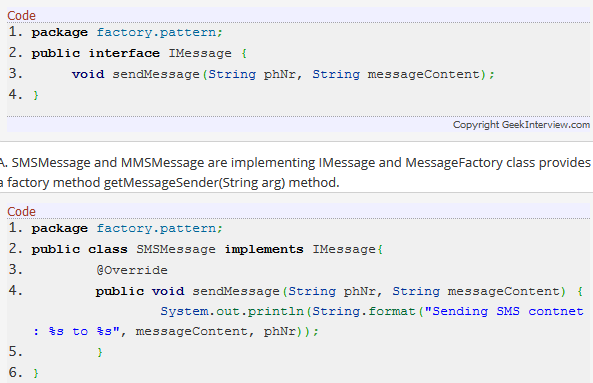
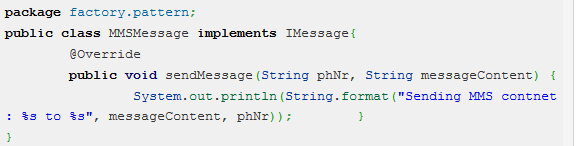
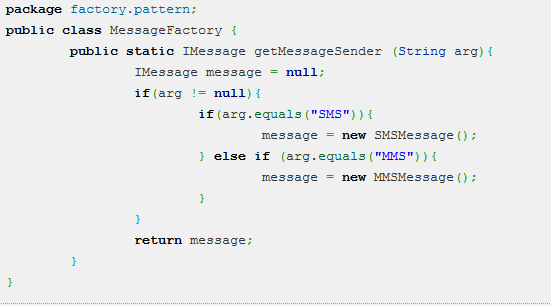
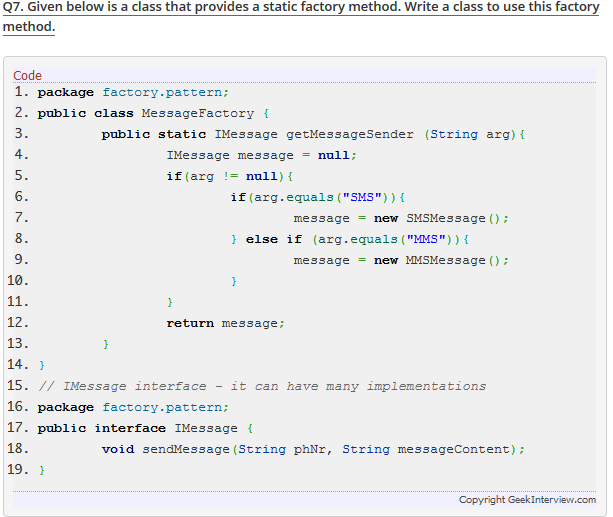
**When will you use a Factory Pattern?**  
A. The factory pattern is preferred in the following cases: - a class does not know which class of objects it must create - factory pattern can be used where we need to create an object of any one of sub-classes depending on the data provided

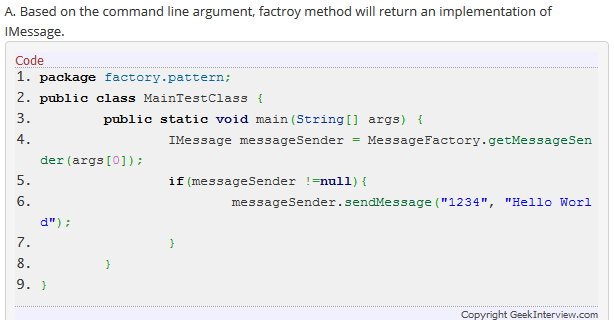
**Give an example of factory method that creates and return an object based on a String parameter. You have following interface. Create 2 implementations for it and write a factory method for it.**

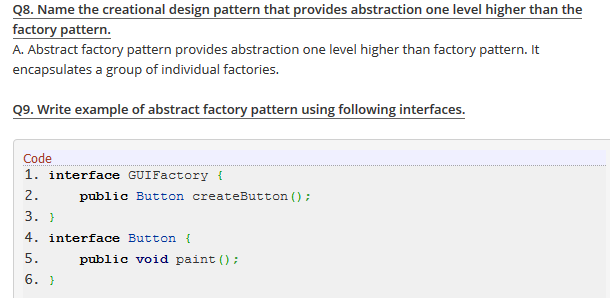




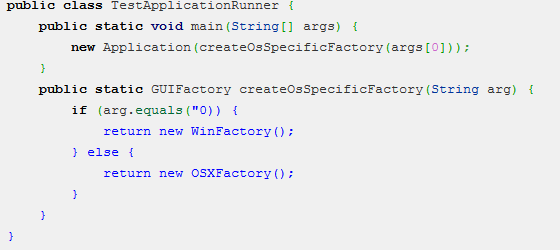












**Q10. What is the difference between factory design pattern and abstract factory design pattern?**  
A. Both are creational design patterns. Factory pattern is said to be there if there is method in a class that is used to create class instances. Any class can have factory methods. Using abstract factory pattern, an instance of a class is created that has only factory methods.

**Q11. When will you use Abstract factory pattern?**  
A. Abstract factory pattern is preferred in situations where the names of actual implementing classes are not needed to be known at the client side. This would mean that we can change the implementation from one factory to another.

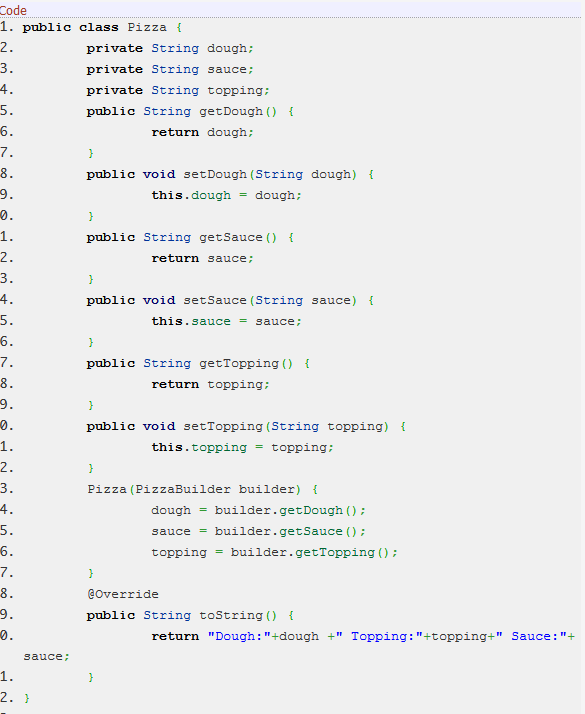
**Q12. Which of the following relation can hold between abstract factory and factory method?  
IS-A  
HAS\_A**  
A. Abstract factory HAS\_A (not always) factory method.

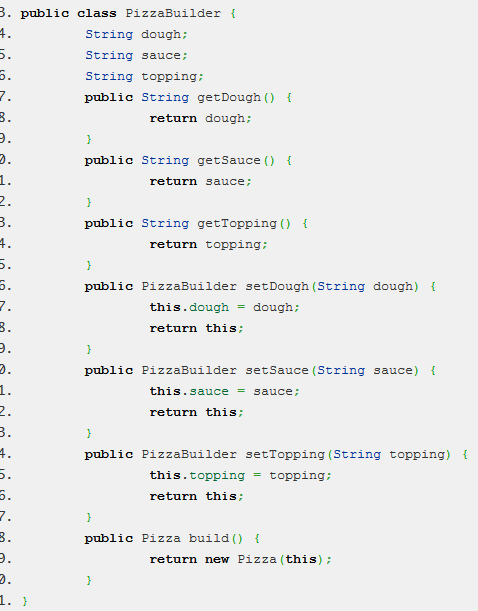
**Q13. What benefits you achieve with factory method?**  
A. Factory method makes the code more flexible and reusable by eliminating the instantiation of application-specific classes. This way the code deals only with the interface and can work with any concrete class that supports this interface.

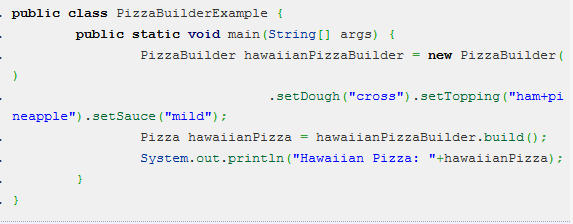
**Q14. What is the liability of using factory method for object creation?**  
A. We need to do subclassing just to instantiate a particular class.

**Q15. Name the design pattern in which a class delegates the responsibility of object instantiation to another object via composition.**  
A. Abstract factory pattern

**Write an example to show how builder pattern can be applied.**  
A. The following example uses builder pattern to construct a Pizza object using PizzaBuilder.

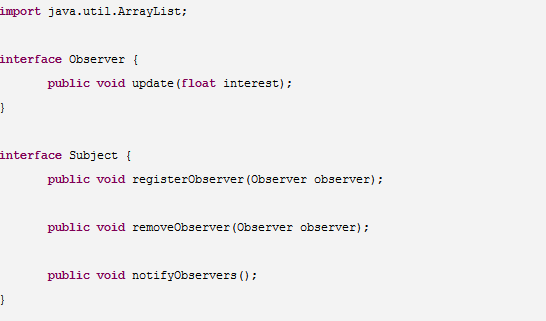






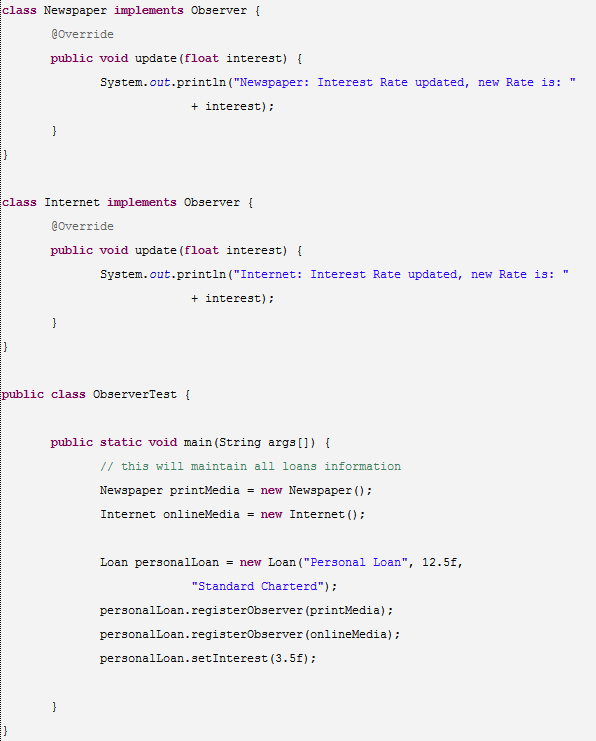
1. **When to use Strategy Design Pattern in Java?**  
   [Java design pattern interview question and answers for senior and experience programmer](http://3.bp.blogspot.com/-1lzFJzIgaHk/UF2Ci6kY5pI/AAAAAAAAAes/OYiM7r-DHzc/s1600/17.jpg)Strategy pattern in quite useful for implementing set of related algorithms e.g. compression algorithms, filtering strategies etc. Strategy design pattern allows you to create Context classes, which uses Strategy implementation classes for applying business rules. This pattern follows open closed design principle and quite useful in Java.  
     
   One of a good example of Strategy pattern from JDK itself is a Collections.sort() method and [Comparator interface](http://java67.blogspot.sg/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html), which is a strategy interface and defines a strategy for comparing objects. Because of this pattern, we don't need to modify sort() method (closed for modification) to compare any object, at the same time we can implement Comparator interface to define new comparing strategy (open for extension).

**2. What is Observer design pattern in Java? When do you use Observer pattern in Java?**  
This is one of the most common Java design pattern interview questions. Observer pattern is based upon notification, there are two kinds of object Subject and Observer. Whenever there is change on subject's state observer will receive notification. See [What is Observer design pattern in Java with real life example](http://javarevisited.blogspot.sg/2011/12/observer-design-pattern-java-example.html) for more details.









**Advantage of Observer Design Pattern in Java:**

Main advantage is **loose coupling** between objects called observer and observable. The subject only know the list of observers it don’t care about how they have their implementation.All the observers are notified by subject in a single event call as **Broadcast communication**

**Disadvantage of Observer Design Pattern in Java:**

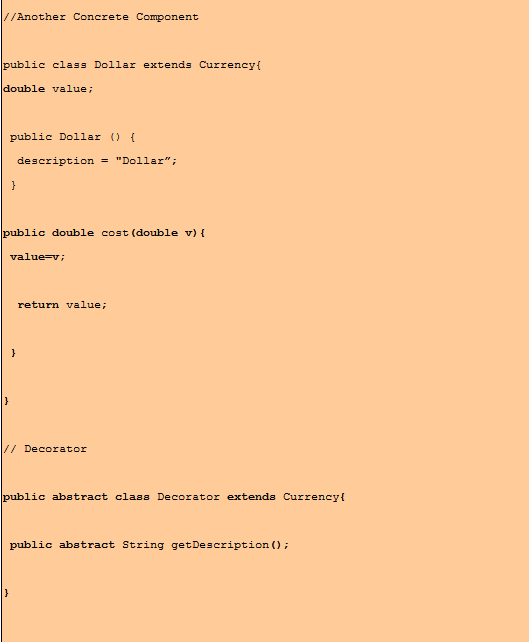
          The disadvantage is that the sometime if any problem comes, [debugging](http://javarevisited.blogspot.com/2011/07/java-debugging-tutorial-example-tips.html) becomes very difficult because flow of control is implicitly between **observers** and **observable** we can predict that now observer is going to fire and if there is chain between observers then debugging become more complex.

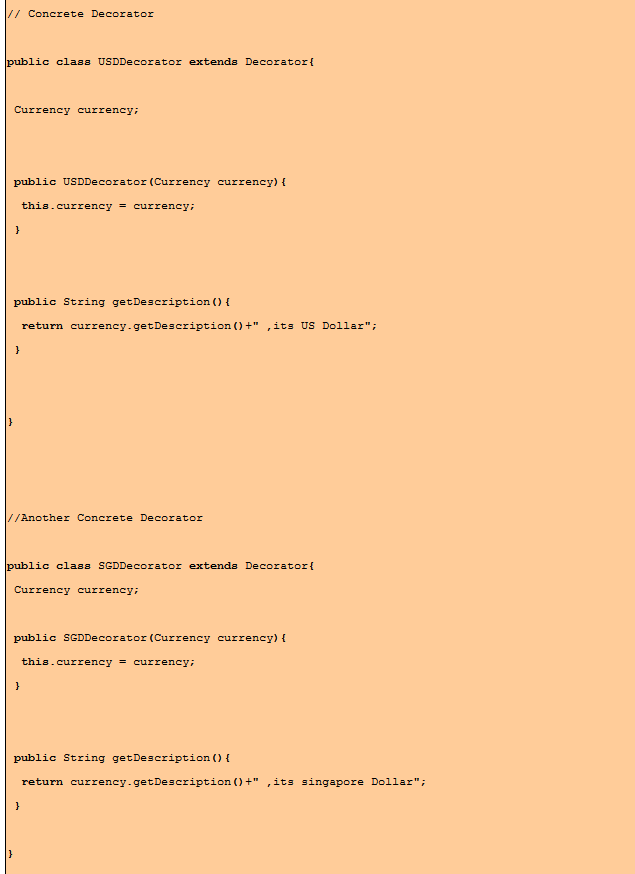
          Another issue is Memory management because subject will hold all the reference of all the observers if we not unregister the object it can create the memory issue.

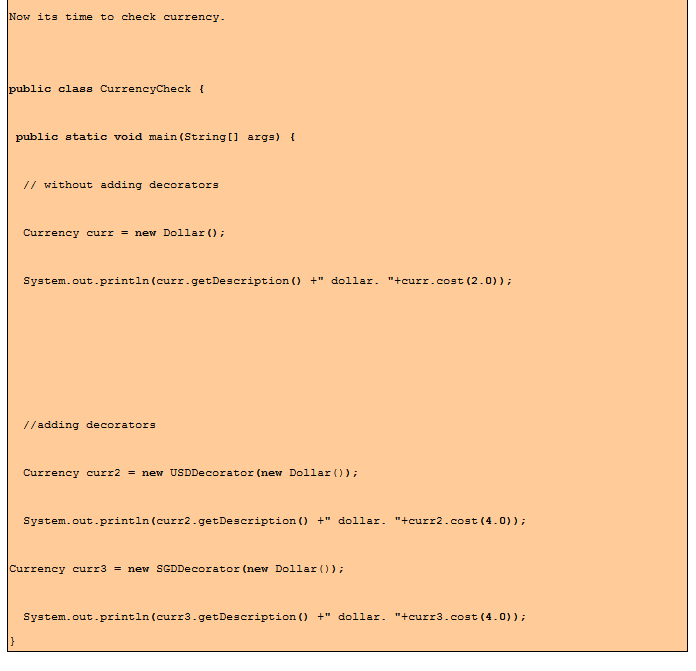
**Difference between Strategy and State design Pattern in Java?**  
This is an interesting Java design pattern interview questions as both Strategy and State pattern has the same structure. If you look at UML class diagram for both patterns they look exactly same, but their intent is totally different.  
  
State design pattern is used to define and manage the state of an object, while Strategy pattern is used to define a set of an interchangeable algorithm and let's client choose one of them. So [Strategy pattern](http://www.shareasale.com/m-pr.cfm?merchantID=53701&userID=880419&productID=546412275) is a client driven pattern while Object can manage their state itself.

**What is decorator pattern in Java? Can you give an example of Decorator pattern?**  
Decorator pattern is another popular Java design pattern question which is common because of its heavy usage in java.io package. BufferedReader and BufferedWriter are a good example of decorator pattern in Java. See [How to use Decorator pattern in Java](http://javarevisited.blogspot.com/2011/11/decorator-design-pattern-java-example.html) for more details.









**Explanation of the code**:

We can understand this in following term;

1.      **Component Interface**: In our example Currency interface is component which used on its own or we need decorator for that.

2.      **Concrete Component: it** implements Component and we add new behavior to this object at dynamically. Dollar and Rupee are the concrete implementation of currency.

3.      **Decorator: Decorator** contains a HAS a Relationship in simple word we can say it has a instance variable that holds reference for component they implement same component which they are going to decorate. Here a Decorator is an abstract class which extends the currency.

4.      **Concrete Decorator:** it’s an implementation of Decorator So USD Dollar and SGD Dollar are the implementation of Decorator contains instance variable for component interface or the thing which they are going to decorate.

**Advantage of Decorator design Pattern in Java**

In brief we see what the main advantages of using decorator design patterns are.

1.      Decorator Pattern is flexible than inheritance because inheritance add responsibilities at compile time and it will add at run-time.

2.      Decorator pattern enhance or modify the object functionality

**Disadvantage**

Main disadvantage of using Decorator Pattern in Java is that the code maintenance can be a problem as it provides a lot of similar kind of small objects (each decorator).

**When to use Composite design Pattern in Java? Have you used previously in your project?**  
This design pattern question is asked on Java interview not just to check familiarity with the Composite pattern but also, whether a candidate has the real life experience or not.  
The *Composite pattern* is also a core Java design pattern, which allows you to treat both whole and part object to treat in a similar way. Client code, which deals with a Composite or individual object doesn't differentiate between them, it is possible because Composite class also implement the same interface as their individual part.  
  
One of the good examples of the Composite pattern from JDK is JPanel class, which is both Component and Container.  When the paint() method is called on JPanel, it internally called paint() method of individual components and let them draw themselves.  
  
On the second part of this design pattern interview question, be truthful, if you have used then say yes, otherwise say that you are familiar with the concept and used it by your own. By the way, always remember, giving an example from your project creates a better impression.

**What is Singleton pattern in Java?**  
Singleton pattern in Java is a pattern which allows only one instance of Singleton class available in the whole application. java.lang.Runtime is a good example of Singleton pattern in Java. There are lot's of follow up questions on Singleton pattern see [10 Java singleton interview question answers](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html) for those followups

### What is double checked locking in Singleton?

One of the most hyped question on Singleton pattern and really demands complete understanding to get it right because of Java Memory model caveat prior to Java 5. If a guy comes up with a solution of using [volatile keyword](http://javarevisited.blogspot.sg/2012/03/difference-between-transient-and.html) with Singleton instance and explains it then it really shows it has in depth knowledge of Java memory model and he is constantly updating his Java knowledge.  
  
Answer : Double checked locking is a technique to prevent creating another instance of Singleton when call to getInstance() method is made in multi-threading environment. In Double checked locking pattern as shown in below example, singleton instance is checked two times before initialization. See [here](http://javarevisited.blogspot.sg/2014/05/double-checked-locking-on-singleton-in-java.html) to learn more about double-checked-locking in Java.

public static **Singleton** getInstance(){

**if**(**\_INSTANCE** == **null**){

synchronized(**Singleton**.class){

//double checked locking - because second check of Singleton instance with lock

**if**(**\_INSTANCE** == **null**){

**\_INSTANCE** **=** **new** **Singleton**();

}

}

}

**return** **\_INSTANCE**;

}

### How do you prevent for creating another instance of Singleton using clone() method?

This type of questions generally comes some time by asking how to break singleton or when Singleton is not Singleton in Java.

Answer : Preferred way is not to implement Cloneable interface as why should one wants to create clone() of Singleton and if you do just throw Exception from clone() method as “Can not create clone of Singleton class”.

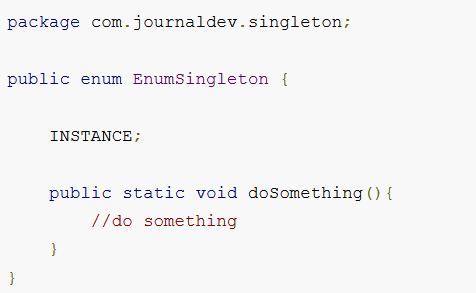
### How do you prevent for creating another instance of Singleton using reflection?

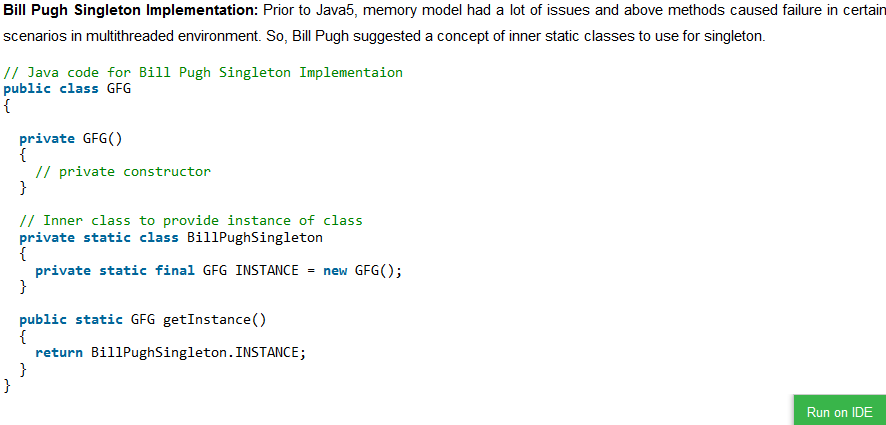
Open to all. In my opinion throwing exception from constructor is an option.  
Answer: This is similar to previous interview question. Since constructor of Singleton class is supposed to be private it prevents creating instance of Singleton from outside but [Reflection can access private fields and methods](http://javarevisited.blogspot.sg/2012/05/how-to-access-private-field-and-method.html), which opens a threat of another instance. This can be avoided by throwing Exception from constructor as “Singleton already initialized”

**How do you prevent for creating another instance of Singleton during serialization?**

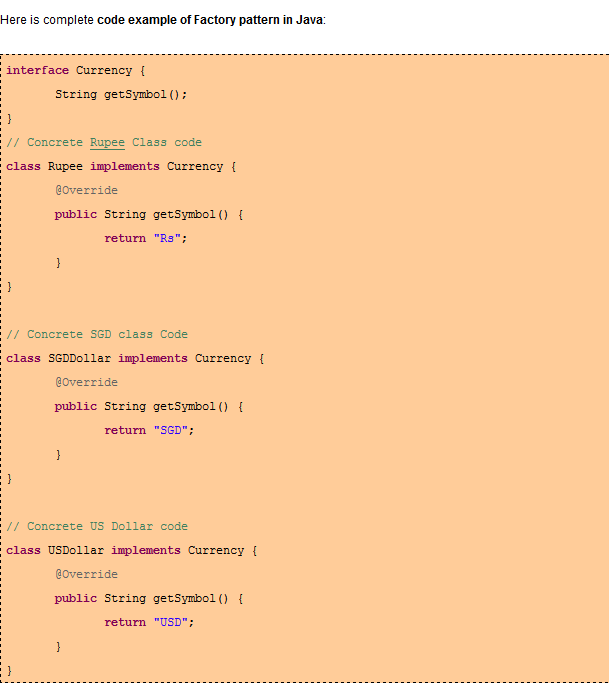
Another great question which requires knowledge of [Serialization in Java](http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html) and how to use it for persisting Singleton classes. This is open to you all but in my opinion use of readResolve() method can sort this out for you.  
Answer: You can prevent this by using readResolve() method, since during serialization readObject() is used to create instance and it return new instance every time but by using readResolve you can replace it with original Singleton instance. I have shared code on how to do it in my post Enum as Singleton in Java. This is also one of the reason I have said that use Enum to create Singleton because serialization of enum is taken care by JVM and it provides guaranteed of that.

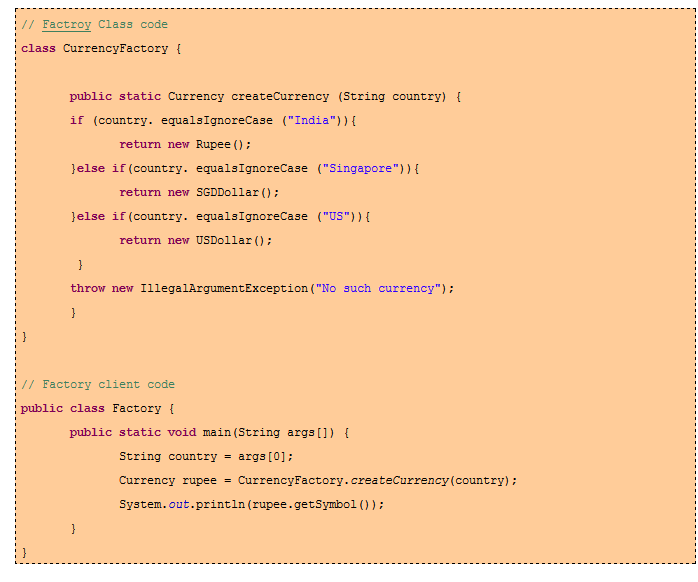
Enum Singleton

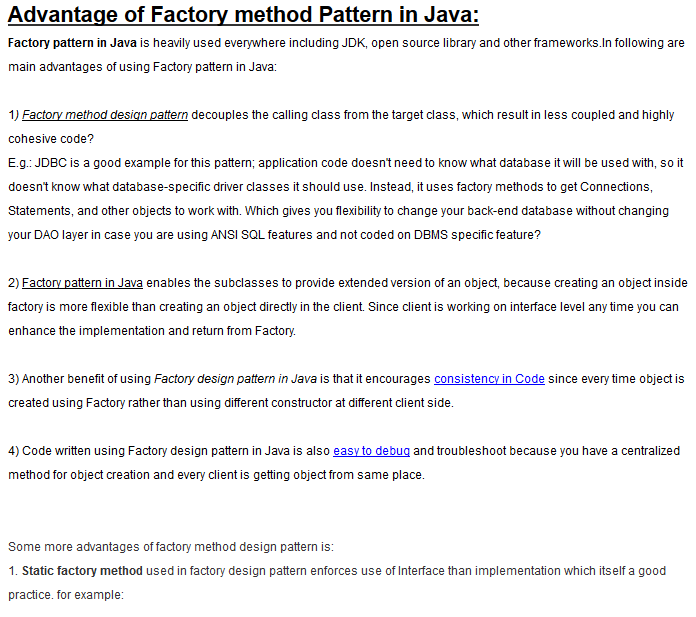




**When to use Template method design Pattern in Java?**  
The Template pattern is another popular core Java design pattern interview question. I have seen it appear many times in real life project itself. Template pattern outlines an algorithm in form of template method and lets subclass implement individual steps.  
  
The key point to mention, while answering this question is that template method should be final, so that subclass can not override and change steps of the algorithm, but same time individual step should be abstract, so that child classes can implement them.  
  
  
**9 What is Factory pattern in Java? What is the advantage of using a static factory method to create an object?**  
Factory pattern in Java is a creation Java design pattern and favorite on many Java interviews.Factory pattern used to create an object by providing static factory methods. There are many advantages of providing factory methods e.g. caching immutable objects, easy to introduce new objects etc. See [What is Factory pattern in Java and benefits](http://javarevisited.blogspot.sg/2011/12/factory-design-pattern-java-example.html) for more details.





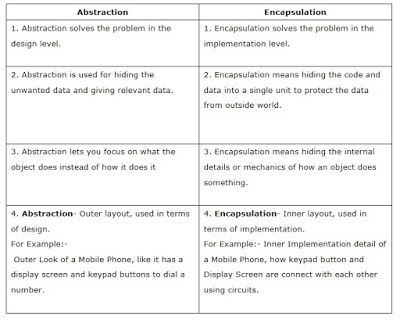


**What is the difference between Decorator and Proxy pattern in Java?**  
Another tricky Java design pattern question and trick here is that both Decorator and Proxy implements the interface of the object they decorate or encapsulate. As I said, many Java design pattern can have similar or exactly same structure but they differ in their intent.  
  
Decorator pattern is used to implement functionality on an already created object, while a Proxy pattern is used for controlling access to an object.  
  
One more difference between Decorator and the Proxy design pattern is that Decorator doesn't create an object, instead, it get the object in its constructor, while Proxy actually creates objects. You can also read [Head First Analysis and Design](http://aax-us-east.amazon-adsystem.com/x/c/QpA7t74JPKiY61JynBRROvsAAAFhHlOCQQEAAAFKAT1A-cM/http:/www.amazon.com/dp/0596008678/ref=as_at?creativeASIN=0596008678&linkCode=w61&imprToken=J1RQa3oyIDrtmu4HSN0erQ&slotNum=0&tag=javamysqlanta-20) to understand the difference between them.  
  
  
**11. When to use Setter and Constructor Injection in Dependency Injection pattern?**  
Use Setter injection to provide optional dependencies of an object, while use Constructor iInjection to provide a mandatory dependency of an object, without which it can not work. This question is related to [Dependency Injection design pattern](http://javarevisited.blogspot.com/2012/12/inversion-of-control-dependency-injection-design-pattern-spring-example-tutorial.html) and mostly asked in the context of Spring framework, which is now become a standard for developing Java application.  
  
Since Spring provides IOC container, it also gives you a way to specify dependencies either by using setter methods or constructors. You can also take a look my [previous post](http://javarevisited.blogspot.com/2012/11/difference-between-setter-injection-vs-constructor-injection-spring-framework.html) on the same topic.  
  
  
**12. What is difference between Factory and Abstract Factory in Java**  
I have already answered this question in detail with my article with the same title. The main difference is that Abstract Factory creates factory while Factory pattern creates objects. So both abstract the creation logic but one abstract is for factory and other for items. You can see [here](http://javarevisited.blogspot.sg/2013/01/difference-between-factory-and-abstract-factory-design-pattern-java.html) to answer this Java design pattern interview question.  
  
  
**13. When to use Adapter pattern in Java? Have you used it before in your project?**  
Use Adapter pattern when you need to make two class work with incompatible interfaces. Adapter pattern can also be used to encapsulate third party code so that your application only depends upon Adapter, which can adapt itself when third party code changes or you moved to a different third party library.  
  
By the way, this Java design pattern question can also be asked by providing an actual scenario. You can further read [Head First Design Pattern](http://aax-us-east.amazon-adsystem.com/x/c/QpA7t74JPKiY61JynBRROvsAAAFhHlOCQQEAAAFKAT1A-cM/http:/www.amazon.com/dp/0596007124/ref=as_at?creativeASIN=0596007124&linkCode=w61&imprToken=J1RQa3oyIDrtmu4HSN0erQ&slotNum=1&tag=javamysqlanta-20) to learn more about Adapter pattern and its real world usage. The book is updated for Java 8 as well so you will learn new, Java 8 way to implement these old design patterns.

**Can you write code to implement producer consumer design pattern in Java?**  
The Producer-consumer design pattern is a concurrency design pattern in Java which can be implemented using multiple ways. If you are working in Java 5 then its better to use Concurrency util to implement producer-consumer pattern instead of plain old [wait and notify in Java](http://javarevisited.blogspot.sg/2011/05/wait-notify-and-notifyall-in-java.html).  Here is a good example of implementing [producer consumer problem using BlockingQueue in Java](http://javarevisited.blogspot.sg/2012/02/producer-consumer-design-pattern-with.html).  
  
  
**15. What is Open closed design principle in Java?**  
The Open closed design principle is one of the SOLID principle defined by Robert C. Martin, popularly known as Uncle Bob in his most popular book, [Clean Code](http://aax-us-east.amazon-adsystem.com/x/c/QpA7t74JPKiY61JynBRROvsAAAFhHlOCQQEAAAFKAT1A-cM/http:/www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=as_at?creativeASIN=0132350882&linkCode=w61&imprToken=J1RQa3oyIDrtmu4HSN0erQ&slotNum=3&tag=javamysqlanta-20). This principle advises that a code should be open for extension but closed for modification.

At first, this may look conflicting but once you explore the power of polymorphism, you will start finding patterns which can provide stability and flexibility of this principle.  
  
One of the key examples of this is State and Strategy design pattern, where Context class is closed for modification and new functionality is provided by writing new code by implementing a new state of strategy. See [this](http://javarevisited.blogspot.com/2011/11/great-example-of-open-closed-design.html) article to know more about Open closed principle.  
  
  
**16. What is Builder design pattern in Java? When do you use Builder pattern?**  
Builder pattern in Java is another creational design pattern in Java and often asked in Java interviews because of its specific use when you need to build an object which requires multiple properties some optional and some mandatory. See [When to use Builder pattern in Java](http://javarevisited.blogspot.sg/2012/06/builder-design-pattern-in-java-example.html) for more details  
  
  
**17. Can you give an example of  SOLID design principles in Java?**  
There are lots of SOLID design pattern which forms acronym SOLID e.g.  
1. Single Responsibility Principle or SRP  
3. Open Closed Design Principle or OCD  
3. Liskov Substitution Principle  
4. Interface Segregation Principle  
5. Dependency Inversion Principle.

**What is the difference between Abstraction and Encapsulation in Java?**  
Even though both Abstraction and Encapsulation looks similar because both hide complexity and make the external interface simpler there is a subtle difference between them. Abstraction hides logical complexity while Encapsulation hides Physical Complexity.  
  
Btw, I have already covered answer of this Java interview question in my previous post as [Difference between encapsulation and abstraction in Java](http://java67.blogspot.sg/2012/08/difference-between-abstraction-and-encapsulation-java-oops.html), here are some more difference from that post.

[](http://www.shareasale.com/m-pr.cfm?merchantID=53701&userID=880419&productID=546412145)

**S.O.L.I.D stands for:**

When expanded the acronyms might seem complicated, but they are pretty simple to grasp.

* **S** - Single-responsiblity principle
* **O** - Open-closed principle
* **L** - Liskov substitution principle
* **I** - Interface segregation principle
* **D** - Dependency Inversion Principle

Let's look at each principle individually to understand why S.O.L.I.D can help make us better developers.

## [Single-responsibility Principle](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-single-responsibility-principle)

**S.R.P** for short - this principle states that:

A class should have one and only one reason to change, meaning that a class should have only one job.

For example, say we have some shapes and we wanted to sum all the areas of the shapes. Well this is pretty simple right?

## [Open-closed Principle](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-open-closed-principle)

Objects or entities should be open for extension, but closed for modification.

## [Liskov substitution principle](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-liskov-substitution-principle)

Let **q(x)** be a property provable about objects of **x** of type **T**. Then **q(y)** should be provable for objects **y** of type **S** where **S** is a subtype of **T**.

All this is stating is that every subclass/derived class should be substitutable for their base/parent class.

## [Interface segregation principle](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-interface-segregation-principle)

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

## [Dependency Inversion principle](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-dependency-inversion-principle)

The last, but definitely not the least states that:

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

This might sound bloated, but it is really easy to understand. This principle allows for decoupling, an example that seems like the best way to explain this principle:

## [Conclusion](https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#toc-conclusion)

Honestly, **S.O.L.I.D** might seem to be a handful at first, but with continuous usage and adherence to its guidelines, it becomes a part of you and your code which can easily be extended, modified, tested, and refactored without any problems.